Scientific Programming 17 (2009) 135–151
DOI 10.3233/SPR-2009-0273
IOS Press

Implementation of scientific computing applications on the Cell Broadband Engine

Guochun Shi [a], Volodymyr V. Kindratenko [a,*], Ivan S. Ufimtsev [b], Todd J. Martinez [b], James C. Phillips [c] and Steven A. Gottlieb [d]

[a] *National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, Urbana, IL, USA*
[b] *Department of Chemistry, University of Illinois at Urbana-Champaign, Urbana, IL, USA*
[c] *Theoretical and Computational Biophysics Group, Beckman Institute, University of Illinois at Urbana-Champaign, Urbana, IL, USA*
[d] *Department of Physics, Indiana University, Bloomington, IN, USA*

Abstract. The Cell Broadband Engine architecture is a revolutionary processor architecture well suited for many scientific codes. This paper reports on an effort to implement several traditional high-performance scientific computing applications on the Cell Broadband Engine processor, including molecular dynamics, quantum chromodynamics and quantum chemistry codes. The paper discusses data and code restructuring strategies necessary to adapt the applications to the intrinsic properties of the Cell processor and demonstrates performance improvements achieved on the Cell architecture. It concludes with the lessons learned and provides practical recommendations on optimization techniques that are believed to be most appropriate.

Keywords: Cell Broadband Engine, molecular dynamics, NAMD, quantum chromodynamics, MILC, quantum chemistry, direct SCF, ERIs

1. Introduction

In this paper, three case studies are presented in which computationally demanding applications are implemented on the Cell Broadband Engine (Cell/B.E.) processor. The applications chosen for the Cell/B.E. implementation are classic examples of the high-performance computing (HPC) codes that are typically executed on large-scale parallel systems. They include:

- Nanoscale Molecular Dynamics (NAMD) [15]. The NAMD SPEC 2006 CPU benchmark [12] code is used as the base for the Cell/B.E. implementation. It is derived from the data layout and inner loop of NAMD to form a compact benchmark for SPEC CPU2006, a CPU-intensive benchmark suite. NAMD belongs to a class of applications that utilize N-body methods [2].
- MIMD Lattice Computation (MILC) [22]. A MILC application that performs simulations with dynamical clover fermions (clover_dynamical) using the hybrid-molecular dynamics R algorithm [8] (su3_rmd) as implemented in MILC version 7.4.0 is considered in this work. MILC belongs to a class of applications that are based on structured grids [2] where the majority of computations on the grid points are vector algebra operations.
- Direct self-consistent field (SCF) method quantum chemistry code. The reference implementation [23] of the two-electron repulsion integrals (ERIs) evaluation code, which is the computational core of the direct SCF method, is written from scratch following the well-known algorithms from the General Atomic and Molecular Electronic Structure System (GAMESS) [18] and other *ab initio* quantum chemistry packages. The ERI kernel is an example of a problem whose computational complexity is O(N^4).

Some of the results described in this paper have been presented in conference papers [19,20] while the quantum chemistry code results are new.

The paper is organized as follows: Section 2 provides a brief description of the Cell/B.E. processor ar-

*Corresponding author: Volodymyr Kindratenko, NCSA, UIUC, 1205 W. Clark St., Urbana, IL 61801, USA. Tel.: +1 217 265 0209; Fax: +1 217 244 1987; E-mail: kindr@ncsa.uiuc.edu.

chitecture. Section 3 discusses related work. Section 4 presents an overview of the computational methods used in the applications and Section 5 gives a detailed account of Cell/B.E. implementations of these applications. Performance results are provided in Section 6 followed by a discussion of various implementation issues and lessons learned.

2. Cell/B.E. architecture overview

The Cell/B.E. system used in this study is a 3.2 GHz dual-Cell blade QS20 server. Two Cell/B.E. processors are used in the blade, and the applications can seamlessly use both of them. The system runs Fedora Core 7 Linux OS with kernel 2.6.22-BSC and the IBM SDK for Multicore Acceleration ver. 3.0.

The Cell/B.E. is a heterogeneous system consisting of one 64-bit PowerPC core called the Power Processor Element (PPE), eight Synergistic Processor Elements (SPEs), system memory, and an I/O controller (Fig. 1). The processing elements are linked by an internal high-speed bus called the Element Interconnect Bus (EIB). The PPE is a 64-bit Power-Architecture-compliant core with 32-kB first-level instruction and data caches and a 512-kB second-level cache. Each SPE consists of a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC), which includes a DMA controller, a Memory Management Unit (MMU), a bus interface, and an atomic unit for synchronization with other SPEs and PPE. SPU is a Single Instruction, Multiple Data (SIMD) processor whose load and store instructions are performed in local address space.

The SPU has two execution pipelines: The floating-point and fixed-point units are on the even pipeline while the rest of the functional units are on the odd pipeline. The SPU can issue and complete up to two instructions per cycle, one on each execution pipeline. Simple fixed-point operations take two cycles, and single-precision floating-point and load instructions take six cycles. Two-way SIMD double-precision floating-point is also supported, but the maximum issue rate is one SIMD instruction per seven cycles.

For the 3.2 GHz Cell/B.E., the EIB is capable of providing peak bandwidth of 204.8 GB/s. The memory interface controller provides 25.6 GB/s to system memory. The I/O controller provides peak bandwidths of 25 GB/s inbound and 35 GB/s outbound. The eight SPUs of the Cell/B.E. processor have combined theoretical peak performance of 204.8 GFLOPS in single precision and 14.63 GFLOPS in double precision.

3. Related work

Several research efforts have been reported detailing molecular dynamics (MD) code ports to the Cell/B.E. platform. Oliver et al. [14] describe a 2.4 GHz Cell/B.E. implementation of a GROMACS kernel optimized for calculating interactions between water molecules. Kernel-only speedup using eight SPEs versus a single-core 3.4 GHz Intel Xeon processor is 15.5×; it is only nearly 2× as fast as PPC 970 running a hand-

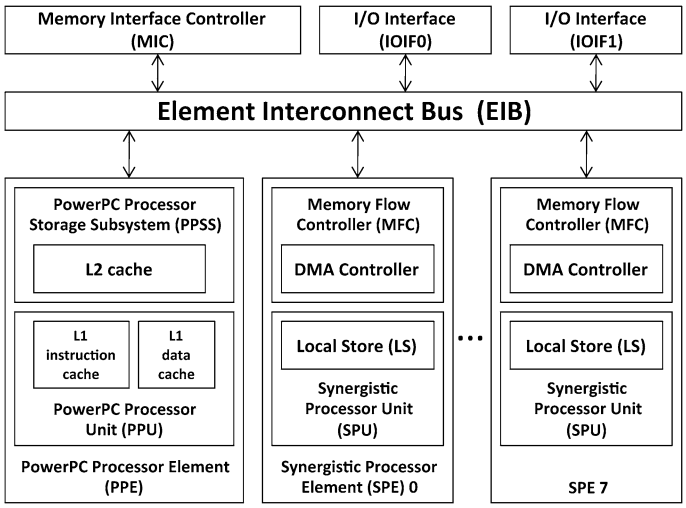

Fig. 1. Cell/B.E. processor architecture.

tuned VMX code. Meredith et al. [10] report on an implementation of a simplified MD simulation code on the Cell/B.E. processor as well as on the GPU and MTA-2 platforms. The kernel speedup achieved on the Cell/B.E. processor is 5× as compared to the code executed on a 2.2 GHz AMD Opteron processor. De Fabritiis [6] reports on a 3.2 GHz Cell blade implementation of a short-range non-bonded force field that computes both electrostatic and L–J interactions using single-precision floating-point arithmetic. Overall code speedup, as compared to the same code optimized for a 2.0 GHz AMD Opteron processor, is 19×.

Bilardi et al. [4] describe a theoretical approach to analyze tradeoffs between bandwidth, memory, and processing for lattice QCD computations and provide quantitative guidelines for several architectures, including the Cell/B.E. processor. They are first to point out the potential of the Cell/B.E. processor for lattice QCD applications. Belletti et al. [3] propose a performance model of a lattice QCD kernel on the Enhanced Cell/B.E. processor and investigate several possible data layouts. The authors conclude that sustained performance on the order of 20% of the theoretical peak performance can be obtained on large machines. Motoki and Nakamura [11] report on an attempt to port matrix–vector multiplication calculations appearing in the fermion conjugate gradient solver to the Cell/B.E. processor from the LTKf90 QCD code written in Fortran 90. The authors report achieving 5% of the theoretical peak performance on a dual-Cell blade. Wolf [24] reports on the implementation of primitives, such as complex numbers and SU(3) matrices, used in QCD applications as C++ class methods targeted for the execution on the Cell/B.E. processor. Various approaches for implementing parts of the QDP++ lattice QCD library on the Cell/B.E. processor are explored by Spray [21]. In particular, the author reports 41 GFLOPS obtained on a single Cell/B.E. processor for the Wilson Dslash operator. Pleiter [16] examines performance-critical kernels in a lattice QCD application, such as linear algebra operations and sparse matrix–vector multiplication, and provides benchmark results for memory access operations.

Ramdas et al. [17] discuss how extrinsic vectorization may be unified with shell structure in ERI calculations through the exploitation of memory access locality. The authors conclude that one should be able to take advantage of both extrinsic vectorization and shell structure value reuse by generating ERIs according to shell structure order and then reordering ERIs into sets of matching class. No actual implementation or performance results have been presented. Hayashi et al. [9] describe the Cell/B.E. implementation of the two electron integral calculations. The results indicate that the performance of the Cell/B.E. implementation of the algorithm running in the Cell/B.E. simulator is not as good as the performance of the same algorithm implemented on a 3.2 GHz Pentium D processor.

Our work differs in that we attempt to implement complete applications and investigate Cell blade performance rather than the performance of a single SPU or single Cell/B.E. processor.

4. Applications and algorithms

4.1. Molecular dynamics

The computational core of any molecular dynamics code is the non-bonded force-field calculation that involves computing the van der Waal's forces – approximated by the Lennard–Jones (L–J) 6–12 potential – and electrostatic (Coulomb) interactions between the non-bonded atom pairs:

$$U_{\text{vdW}} = \sum_i \sum_{j>i} 4\varepsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right],$$

$$U_{\text{Coulomb}} = \sum_i \sum_{j>i} \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}}.$$

In theory, these forces need to be computed between all pairs of non-bonded atoms and are then applied to move the atoms according to the Newtonian equation of motion. In practice, MD programs, such as NAMD, apply multiple optimization techniques to reduce the time needed to compute the forces. The entire simulation space is divided into 3D cells, called patches, whose size is related to the cutoff radius beyond which no interaction calculations are performed. The atoms from each patch are checked against themselves and against the atoms from the 13 neighbor patches (Newton's third law of motion is applied here to eliminate redundant calculations) and the corresponding interaction calculations are performed on these patches. Cutoff radius is applied to limit the calculations to only those atoms that are close to each other. The smooth particle-mesh Ewald (SPME) method is used for full electrostatic computations with the direct component of PME sum substituting the Coulomb equation. Interpolation tables are used for both L–J potential and Coulomb.

4.2. Lattice quantum chromodynamics

Lattice Quantum Chromodynamics (QCD) is a quantum field theory of the strong interaction formulated on a discretized space-time. The quarks are described by fields that have three components, called colors. The gluons, which are the force carriers and are analogous to photons in quantum electrodynamics, are described by matrix valued fields. Each matrix is a 3×3 unitary matrix with determinant 1, that is, an element of SU(3). At each space-time grid point, there is one such gauge matrix "pointing" to each neighboring grid point.

There are a number of ways to formulate the quarks on the lattice. In this work, simulations with dynamical clover quarks using the hybrid-molecular dynamics R algorithm (su3_rmd) [8] as implemented in the MILC version 7.4.0 code base are considered. In the clover formulation, the quarks have a Dirac spinor index that can take the values 1–4. Thus, at each site of the lattice the clover quark field has three color and four spinor components for a total of 12 complex components.

The interaction between the quarks and gluons is described by this term in the Lagrangian:

$$\begin{aligned}&\bar{\psi}(x)M(U)_{xy}\psi(y)\\&\quad\equiv \bar{\psi}(x)\psi(x) + \kappa\bar{\psi}(x)\big[(1+\gamma_\mu)U_{x,\mu}\psi(x+\hat{\mu})\\&\qquad + (1-\gamma_\mu)U^{\dagger}_{x-\hat{\mu},\mu}\psi(x-\hat{\mu})\big]\\&\qquad + c_{\mathrm{SW}}F_{\mu\nu}\bar{\psi}(x)\sigma_{\mu\nu}\psi(x).\end{aligned}$$

In this formula the color indices on the quark fields ψ and $\bar{\psi}$ and the matrices $U_{x,\mu}$ are suppressed. The lattice sites are denoted by x and y, and the directions in space-time by μ and ν. Repeated indices are summed over. The set of four 4×4 matrices γ_μ are known as the Dirac gamma matrices, κ is known as the hopping parameter and is related to the quark mass, $U_{x,\mu}$ is the gauge matrix connecting the sites x and $x + \hat{\mu}$, c_{SW} is a constant, $F_{\mu\nu}$ is the field strength tensor constructed from products of U matrices, and $\sigma_{\mu\nu}$ are proportional to commutators of the Dirac gamma matrices.

A large part of the time in the calculation is spent on inverting the matrix $M(U)$ as the gluon field U evolves.

4.3. Quantum chemistry

The calculation of two-electron repulsion integrals remains the bottleneck in many of the *ab initio* molecular orbital (MO) or density functional theory (DFT) electronic structure codes. For example, in direct self-consistent field methods many millions of electron repulsion integrals are recomputed every SCF iteration and count for the vast majority of the execution time.

The two-electron repulsion integrals over contracted basis functions can be computed as:

$$\begin{aligned}&(\mu\nu|\lambda\sigma)\\&\quad= \sum_{p=1}^{N_\mu}\sum_{q=1}^{N_\nu}\sum_{r=1}^{N_\lambda}\sum_{s=1}^{N_\sigma} d_{\mu p}\, d_{\nu q}\, d_{\lambda r}\, d_{\sigma s}[pq|rs],\end{aligned}$$

where N_* is the contraction length, d_{**} are contraction coefficients, and $[pq|rs]$ are integrals evaluated over primitive basis functions. The Rys quadrature scheme for a two-electron Coulomb repulsion integral is used to evaluate primitive integrals $[pq|rs]$ for Gaussian-type orbitals (GTO) basis sets [7]. In this work only $(ss|ss)$ integrals over contracted s-orbitals are considered. The general formula for primitive $[ss|ss]$ integrals is as follows [5]:

$$\begin{aligned}[s_1s_2|s_3s_4] = \frac{\pi^3}{AB\sqrt{A+B}}K_{12}(\vec{\mathbf{R}}_{12})K_{34}(\vec{\mathbf{R}}_{34})\\ \times F_0\bigg(\frac{AB}{A+B}[\vec{\mathbf{R}}_p - \vec{\mathbf{R}}_Q]^2\bigg),\end{aligned}$$

where

$$\begin{aligned}&A = \alpha_1 + \alpha_2,\\&B = \alpha_3 + \alpha_4,\\&F_0(t) = \frac{\mathrm{erf}(\sqrt{t})}{\sqrt{t}},\\&\vec{\mathbf{R}}_{\mathbf{kl}} = \vec{\mathbf{R}}_{\mathbf{k}} - \vec{\mathbf{R}}_{\mathbf{l}},\\&\vec{\mathbf{R}}_{\mathbf{p}} = \frac{\alpha_1\vec{\mathbf{R}}_{\mathbf{1}} + \alpha_2\vec{\mathbf{R}}_{\mathbf{2}}}{A},\\&\vec{\mathbf{R}}_{\mathbf{Q}} = \frac{\alpha_3\vec{\mathbf{R}}_{\mathbf{3}} + \alpha_4\vec{\mathbf{R}}_{\mathbf{4}}}{B},\\&K_{ij}(\vec{\mathbf{R}}_{ij}) = \exp\bigg(-\frac{\alpha_i\alpha_j}{\alpha_i+\alpha_j}[\vec{\mathbf{R}}_i - \vec{\mathbf{R}}_j]^2\bigg),\end{aligned}$$

α_k is the exponent, and $\vec{\mathbf{R}}_{\mathbf{k}}$ is the atomic center of the kth primitive basis function in the integral.

Microprocessor implementation of the two-electron repulsion integrals evaluation code is straightforward. The four outer loops sequence through all unique com-

binations of electron shells. For each such permutation, the four inner loops sequence through all shell primitives. Inside these four nested loops, primitive $[ss|ss]$ integrals are computed via the above equations and are summed. Gauss error function, erf, is computed via a lookup table interpolation involving just five coefficients. Contracted integrals are stored in memory for follow-up use for constructing the Fock matrix necessary to solve the electronic time-independent Schrodinger equation.

5. Cell/B.E. implementation

5.1. NAMD

While the basic MD algorithm appears to be structurally simple, the actual implementation of the algorithm in NAMD is more complex. Thus, the NAMD code that implements the actual non-bonded force field consists of fewer than 100 lines of C code, while the code that implements pair list and bonded force calculations is almost 500 lines of C code with special provisions made for fixed atoms and atoms from hydrogen groups as well as memory management for pair list data, lookup table pointers, etc. Such an implementation does not lend itself to an efficient port to the Cell/B.E. processor because of its size and complexity with non-compute-intensive operations. Therefore, a restructured kernel is used for the implementation in which no pair list is generated and stored and no bonded force calculations are performed as they are responsible for only a fraction of the overall execution time of the code. Instead, if a pair of atoms from two patches is found to be within a cutoff distance, the non-bonded forces are computed right away. Thus, only the compute-intensive section of the code – short-range non-bonded force calculation (L–J potential and PME direct sum substituting for the Coulomb equation, both implemented using lookup tables) – is considered for Cell/B.E. implementation, and the bonded forces are dealt with outside of the accelerated non-bonded force-field kernel.

In NAMD, each patch typically consists of a few hundred atoms, which is a small enough amount of data to fit into SPE's local store (LS). Therefore, all atoms for one patch (self-compute) or two patches (pair-compute) are loaded into SPE memory at once after a task is started in SPE. Forces are computed, stored in the SPE's local store, and transferred back to the system memory after the patch is processed. Two lookup tables (L–J lookup table and table_four) also fit into SPE's local store and are loaded during the initialization. Techniques for hiding communication latency while doing the computation (e.g., double buffering) are not necessary here as the communication time is typically less than 1% of the overall execution time. This is common for kernels with a substantial data reuse.

An efficient SIMD computation kernel in SPE is the key to successful implementation. NAMD has a single compute kernel that can be used either for self-compute or pair-compute depending on the input parameters. If it is invoked with a single patch of atoms as the input, the self-compute mode is assumed, otherwise, the pair-compute mode is used. These modes of operation have different execution profiles and require different optimization techniques to derive efficient implementations.

As far as the SIMD implementation of the NAMD kernel is concerned, there are two main kernel stages: cutoff distance test and short range force computation which is only performed if the cutoff distance test passes. This kernel structure results in a branch divergence that complicates the use of SIMD instructions. There are two ways to deal with the branch divergence in this case: (1) perform the short range force computation without regards to the cutoff test and discard the results at the end for those cases that fail the cutoff test, and (2) perform the cutoff distance test, but delay the short range force calculations until the number of atom pairs needed to fill in the SIMD instruction is found. The first technique eliminates branch divergence but results in unnecessary computations. The second technique eliminates unnecessary computations but adds data manipulation overhead. Both techniques were found to be applicable in the case of NAMD.

In self-compute mode, about 50% of the time the distance between the pairs of atoms is less than the cutoff distance. Therefore, an efficient way to compute and update the forces for the interacting pairs of atoms is to compute all forces regardless of whether the distance between the considered atoms is less than the cutoff or not. Before updating the final force values, zeros are filled into force elements according to a selection vector generated from comparing the vector of distance and cutoff. In this way, computing forces for individual atom pairs is avoided and the SIMD processing pipeline is fully utilized.

Optimizing the pair-compute kernel is more challenging because the force computations are necessary

in only a small number of cases. In most cases, atoms are found to be outside the cutoff distance from each other and no further calculations are required. Worse is the fact that the atom pairs within the cutoff distance from each other are sparsely distributed among the other atom pairs. Most of the time there are no interacting pairs of atoms and only infrequently are there one or two interacting pair of atoms per any four consecutive atom pairs. In order to overcome this difficulty, it is necessary to save data when there are not enough (fewer than four) interacting pairs of atoms and do computations when four or more valid atom pairs are accumulated.

When implemented as stated above, a single invocation of the force computation requires 291 cycles for the self-compute implementation and 653 cycles for the pair-compute implementation of the kernel. The overhead for accumulating data for the SIMD instruction in the pair-compute kernel is approximately 360 cycles.

The original NAMD kernel is implemented using double-precision floating-point values for distance computation and two lookup tables. Since the theoretical peak performance of the Cell/B.E. processor differs significantly for single-precision and double-precision operations, both the single-precision and double-precision kernels are implemented in this work in order to understand the effects of different numerical types on overall application performance. The basic implementation strategy for the double-precision kernel on the Cell/B.E. is the same as for the single-precision floating-point version, with more memory needed for atom data and lookup tables. The double-precision floating-point code is larger than the single-precision kernel. As with the single-precision kernel, the compute loop iterates with the increment of four because there are a lot of single-precision computations even in the double-precision version of the kernel, and therefore they need to be scheduled four at a time for the SIMD core to be efficiently utilized. However, this means that the operations executed in double-precision have to be manually unrolled two at a time, thus resulting in a large codebase for the double-precision floating-point implementation.

Given a group of patches, NAMD sequentially calls the self-compute kernel for each of the patches and then calls the pair-compute kernel for unique pairs of neighbor patches. In the original NAMD implementation, these two sections of the code are implemented as two independent loops. The self-compute kernels can be safely distributed among the eight SPEs on the Cell/B.E. processor as there is no data dependency between individual patches. However, there is data dependency for different tasks in the pair-compute loop if the pair-compute kernels work on the same patch. Fortunately, there is no required order of execution for the pair-compute kernels; therefore, the data dependency issue for individual patches can be resolved by simply keeping track of the status of each individual patch and scheduling the execution of the pair-compute kernels on the SPEs only for those patches that are currently not processed by any other SPEs. This technique was implemented using a pool of patch pairs to be processed and a pool of SPEs available for pair-compute kernel execution and just looping over the unprocessed patch pairs until all of them are processed, delaying the execution of those with current data dependencies until such dependencies are cleared.

5.2. MILC

When porting the MILC code to the Cell/B.E. processor, the main design goal was to preserve MILC's ability to scale on a large number of compute nodes (in the compute cluster sense). Thus, the resulting design is PPE-centric: the application skeleton (including MPI) is executed on the PPE whereas compute kernels are executed on the SPEs. MPICH2-1.0.5p4 MPI implementation compiled for the PPE is used in this work.

MILC's body consists of many small compute loops (kernels) that iterate over subsets of the 4D space-time lattice, and MPI scatter/gather operations in between. This structure provides the scalability necessary to efficiently execute the application on a large distributed memory system. However, as a result, no single kernel is responsible for more than 20% of the overall execution time. The application is composed of 27 major subroutines (Table 1), some of which consist of a single compute kernel, e.g., mult_su3_an, and some consisting of several kernels with MPI scatter/gather operations in between, e.g., udadu_mu_nu. Just 10 of these subroutines are responsible for about 90% of the overall execution time. However, once they are ported to the SPEs, the overall execution time of the application becomes dominated by the execution time of the remaining 17 subroutines left for execution on the PPE. Therefore, all 27 subroutines need to be ported to the SPEs in order to avoid introducing additional computational overhead on the PPE. Since many of these subroutines consist of multiple compute kernels with MPI

Table 1
su3_rmd clover_dynamical application in MILC is composed of 27 major subroutines

Subroutine	Description	# of kernels in the routine	% of overall runtime
udadu_mu_nu	Compute U dA/dU, part of the fermion force term given mu & nu	8	20.5
dslash_w_site	Dirac operator implementation	4	19.2
su3_mat_copy	Matrix copy	1	18.7
mult_su3_nn	Matrix multiplication	1	11.4
mult_su3_na	The first matrix multiplies the conjugate transpose of the second matrix	1	8.2
mult_this_ldu_site	Multiply Wilson_vector by sigma_mu_nu matrix	1	4.7
udadu_mat_mu_nu	Calculate udadu_mu_nu and multiply by an su3_matrix for a term of the fermion force	8	3.1
single_action	Compute single action when updating the momentum matrices	1	2.5
su3_adjoint	Conjugate transpose of a matrix	1	1.8
General strided gather	Compute addresses and MPI message information for exchanging data with neighbors	1	1.6
f_mu_nu	Compute the F{mu, nu} used in the clover fermion action	4	1.5
scalar_mult_add_wvec	Multiply Wilson vector by a scalar and add to another Wilson vector	1	1.0
scalar_multi_su3_matrix_add	Multiply matrix by a scalar and add to another matrix	1	0.9
d_congrad2_cl	Conjugate gradient for clover fermions	3	0.9
set_neighbor	Compute neighbor addresses within the same node	1	0.6
mult_su3_an	The conjugate transpose of the first matrix multiplies the second matrix	1	0.6
update_u	Update the link matrices by going to the sixth order in the exponential of the momentum matrices	1	0.6
compute_clov	Compute clover term	6	0.6
update_h_cl	Update the momentum matrices	1	0.3
scalar_mult_add_wvec_magsq	Multiply Wilson vector with a scalar and add it to another vector; the squared magnitude of the resulting Wilson vector is computed	1	0.3
realtrace_su3_nn	su3_matrix is multiplied by another su3_matrix. The real trace of the result su3_matrix and the third matrix is computed	1	0.2
add_su3_matrix	Addition of two matrices	1	0.2
wp_shrink	Compute the Wilson projection of a Wilson fermion vector	1	0.1
magsq_wvec	Compute the squared magnitude of a Wilson vector	1	0.1
gauge_action	Compute a gauge action	1	0.1
set_su3_matrix_to_zero	Set all elements of a su3_matrix to zero	1	0.1
Reunitarize	Reunitarize a su3_matrix	1	0.1

Notes: Each such subroutine consists of one to eight compute kernels with MPI scatter-gather operations in between. In total 54 such compute kernels were identified and re-implemented for execution on SPEs.

scatter/gather operations in between, 54 unique kernels were identified for implementation on the SPEs.

For any given kernel, there are three types of input data: elements in a lattice site, elements in contiguous memory (usually a temporary memory region created inside MILC for temporary use), and elements in the neighboring site (neighbors in term of (x, y, z, t); they are not physically adjacent to each other in memory). There are only two types of output data: elements in a lattice site and elements in contiguous memory. Because of the similarities in the data types used by all kernels, a common DMA engine was written to load input data into the SPEs' LS and output data back to the main memory.

The Cell/B.E. processor delivers the best memory-to-local store bandwidth when both source address and destination address are aligned at the 128-bytes boundaries and the amount of data to be transferred is a multiple of 128 bytes. Non-aligned DMA requests run at the half of the bandwidth due to the fact that two bus requests instead of one are needed for each cache line of data. We evaluated several approaches to deal with the data alignment issue. The first approach is to pack/unpack data on the PPE on the fly as needed. Data

distributed across the lattice is packed to a contiguous memory buffer aligned at 128-bytes boundary and the packed data is transferred to the SPE's LS. After computations are done, the results are transferred back to the main memory where they are unpacked by the PPE into the corresponding locations in the lattice. The advantage of this approach is that one can easily ensure the desirable memory alignment. However, a very significant overhead of the data packing/unpacking on the PPE was encountered. This is not surprising since the data have to be transferred several times between the PPE and the main memory over the slow interface.

The second approach was to modify the data layout in the lattice array of site structures by replacing each su3_matrix or fwilson_vector in the site structure with a pointer to the corresponding element located in a contiguous and properly aligned memory segment. With this approach, the need to pack and unpack data before transferring it between the main memory and the SPEs is eliminated since the data is already properly aligned. However, this causes the entire program to slow down significantly because each access to an element now ends up being two memory accesses: one for the pointer and the other for the actual data. Moreover, the code changes to implement this approach are non-trivial.

The third approach, which is used in the final implementation, is to add padding to align the data at the 128-byte boundaries. The lattice data structure is allocated to be 128-bytes aligned, and the most commonly used data structures are padded to 128 bytes. Two data structures are padded for this reason: su3_matrix is changed from a 3×3 matrix to a 4×4 matrix, thus changing the size of the su3_matrix from 72 bytes to 128 bytes, and su3_vector is changed from a vector of three complex variables to a vector of four complex variables. This change also makes one of the other commonly used data structures, fwilson_vector, 128 bytes. The obvious disadvantage of this approach is that more bytes of data have to be transferred between the main memory and the SPEs' LS. However, padding helps both to better use the bandwidth between main memory and local store and to make writing SIMD instructions easier since the data structures are already aligned at 16-byte boundaries.

Data from each site is usually accessed in a strided manner, therefore, simply aligning it at the 128-bytes boundary is not sufficient to achieve the full memory bandwidth. Each Cell blade memory node consists of 16 banks, therefore it is important to ensure that strided memory transactions are distributed between the 16 banks rather than using just a few of them all the time. This is implemented by padding the lattice site data to the nearest odd stride size, which turned out to be 21 in our case, in addition to the 128-byte boundary.

In the case of an $8 \times 8 \times 16 \times 16$ lattice volume, the 54 kernels of interest are called 13,408 times. It is therefore impractical to spawn a new SPE thread each time a new kernel is executed because the SPE thread execution overhead will have an adverse impact on the overall application performance. Fortunately, the kernels are small in terms of the actual lines of code, so they can be bundled in a single library of SPE-resident subroutines. The subroutines are invoked via a thin interface running on the SPE as the main SPE thread. Thus, only one thread per SPE is invoked at the start of the application. Each compute kernel in the original CPU-based code is replaced with a small wrapper subroutine, executed on the PPE, that sets up the task structures specific for each individual kernel, signals the SPEs via mailboxes, and waits for the completion message from all the SPEs. The task structure created by these subroutines is copied to a container padded to a multiple of 128 bytes, and the pointer to the container is sent to the SPE as a mailbox message. Upon receiving a mailbox message, each SPE converts the message to a pointer in the global memory space and fetches the first eight bytes via DMA. The first four bytes provide the unique SPE-resident subroutine identifier and the next four bytes indicate the kernel-specific task structure size. The SPE then transfers the entire kernel-specific structure to its LS and passes the control to the corresponding SPE-resident subroutine. Once the calculations are done, results are transferred back to the main memory and a mailbox message is sent to the PPE indicating the completion of the SPE task. PPE polls for the messages from SPEs and resumes its work once all SPE-based kernels return. In theory, this approach may flood EIB with mailbox read requests from the PPE [1]. An alternative approach is to use the interrupt mailbox to notify the PPE about the completion of the SPE task. Both approaches were evaluated in this work and it was found that the interrupt mailbox approach introduces larger delay, thus slowing down the entire application.

5.3. ERIs kernel

In this study, the evaluation of $(ss|ss)$ integrals over contracted s-orbitals is implemented on the Cell/B.E. processor. In the reference microprocessor implementation some quantities are pre-computed outside the in-

nermost loop in O(N^2) steps and do not count for much of the execution time. Therefore, these computations are left to the PPE. The body of the innermost loop is executed in O(N^4) time and has a high data-reuse rate. Therefore it is natural to implement and execute this part of the application in the SPEs.

Memory requirements to store the pre-computed quantities exceed SPE's local store size. As a result, pre-computed arrays of data are stored in the main system memory. These pre-computed quantities are unique for each pair of electron shells and are brought in the SPE's local store as needed. The SPE's local store is sufficient to keep this reduced amount of data. The input data – which includes an array of coordinates, an array of shell primitives and an array of shells – is less than 32 kB combined. The Gauss error function interpolation table contains 3,608 entries consisting of five single-precision floating-point numbers and is about 71 kB.

Code profiling shows that a large percentage of time is spent computing the Gauss error function via the interpolation table. Vectorizing this part of the code "as is" is inefficient because the number of operands is not a multiple of four. But since there is some room left in the local store, the lookup table can be padded to eight 32-bit values, thus making it much simpler to cast operands into vectors of four values and issue SIMD instructions on them. After padding, the total amount of memory for the lookup table is 113 kB.

The output produced by the SPE kernels is much larger than SPE's LS, typically several megabytes. Therefore, the output data is transferred back to the main memory once the local buffer becomes filled. The data transfer time for the input and output parameters, excluding the pre-computed quantities, is negligible compared to the compute time. Therefore, there is no need for hiding DMA transfer time behind the compute time. However, transferring the pre-computed data from main memory to local store needs to be done quite often and therefore double buffering is used for this particular data transfer. Also, contrary to the reference implementation, for relatively sparse basis sets it turns out to be more efficient to compute the quantities on the fly in the SPE rather than to pre-compute them in the PPE and transfer in as needed. Therefore, both approaches are implemented – pre-compute on the PPE and DMA as needed and compute in the SPE on the fly – and are used based on the degree of contraction of the basis set.

A considerable effort went into analyzing the structure of basis sets and its impact on the performance. Depending on the number of primitives in the basis sets, different optimization structures yield different results and no single optimization strategy works well for all cases. Therefore, we attempt to choose one or another optimization strategy from a set of implemented kernels at runtime. For example, if the number of the iterations in the innermost loop is not a multiple of four, the loop order is switched to find one with a number of iterations that is a multiple of four. If this can be done, then variables related to the innermost loop can be cast to vector pointers and can be referenced directly rather than constructing vectors from individual scalar values. In another example, the two innermost loops were manually unrolled for the case of six primitives to increase the compute density of the kernel and to eliminate some of the inefficiencies due to short loops.

If the number of basis primitives is the same for all shells, the workload is distributed evenly between the SPEs based on the number of reduction elements. However, this is not the case for relatively uncontracted basis sets and therefore a different load-balancing procedure is implemented: compute the total number of integrals to be evaluated and distribute the workload based on it. This case, however, requires more calculations on the PPE side.

6. Results and discussion

6.1. NAMD

For performance evaluation, NAMD Cell/B.E. implementation is executed for a single time step on a representative system of 92,224 atoms with a 12 Å short-range cutoff dataset [15]. This dataset is divided into 144 patches, resulting in 144 calls to the self-compute kernel and 144 $\times$ 13 calls to the pair-compute kernel.

The performance of individual kernels (self- and pair-compute) for both single-precision and double-precision numerical types is given in Table 2. Data transfer time for single-precision and double-precision kernels is almost identical for the self-compute kernel (Table 2). This is because the amount of data that needs to be transferred is small enough and the DMA latency time is dominant. However, for the pair-compute kernel, which requires twice the amount of data as the self-compute kernel, an increase in data transfer time by almost a factor of 1.2 is observed when switching from the single-precision to the double-precision numerical type. This corresponds to the fact that the amount of

Table 2
Performance of self- and pair-compute kernels for single- and double-precision implementations

	Self-compute kernel		Pair-compute kernel	
	Data transfer (s)	Total runtime (s)	Data transfer (s)	Total runtime (s)
Single-precision	0.0016	0.701	0.035	6.429
Double-precision	0.0017	1.314	0.042	11.504

Note: The time is taken from running the entire simulation on just one SPE.

data that is transferred back and forth between the main memory and the SPE for each atom changes from 44 to 56 bytes ($56/44 = 1.27$) when switching from single- to double-precision. (Note that in the utilized NAMD SPEC 2006 CPU benchmark code, force values are stored as 32-bit integers while atom displacements are stored as double-precision floating-point values.)

To get a more precise picture of the work performed by each of the kernels, excluding the DMA data transfer time, the SPU static timing tool, spu_timing, is used. It instruments an SPU assembly file with scheduling, timing, and instruction issue estimates assuming straight, linear execution of the program. The total number of clock cycles spent performing five major types of operations is counted (Fig. 2): SIMD distance and force computations, merge of scalar values to form vectors for SIMD force calculations, and scalar distance and force calculations (usually four pairs of atoms at a time are processed using SIMD instructions; however, if there are fewer than four pairs of atoms left, they are processed one by one using the original scalar code). One can see that in the self-compute kernels the majority of the time is spent on the force computation path, while in pair-compute kernels the majority of the time is spent on the distance computation path.

A more detailed analysis of the data presented in Fig. 2 reveals an increase of more than a factor of two for SIMD distance compute time when switching from single-precision to double-precision. At the same time, actual SIMD force compute time for the double-precision kernel is only a factor of 1.8 slower than the single-precision kernel. As one can see from Fig. 2, in the single-precision kernel only about one-third of the time is spent on actual computations, while in the double-precision kernel about two-thirds of all cycles are computation cycles. Furthermore, not all floating-point operations in the double-precision version require double-precision. In fact, in the full-length computation, there are 21 double-precision operations and 20 single-precision operations. Thus, the expected 14× slowdown of the double-precision kernels is not observed.

Both the single-precision and double-precision kernels scale well for multiple SPEs (Fig. 3). Since there are 144 patches (or 144×13 patch pairs in the pair-compute loop) in the input data, it is quite easy to find dependency-free tasks, and therefore all SPEs are kept busy most of the time and the speedup increases linearly with the addition of more SPEs. However, it is easy to see that this trend may not hold if the dataset is divided into only a small number of patches or a large number of SPEs are used since a dependency-free task may not be readily available at any given moment.

6.2. MILC

The 54 computational kernels that were ported to the Cell/B.E. processor are responsible for 98.8% of the overall execution time on the 2.3 GHz Intel Xeon chip; only 1.2% of the overall execution time is due to the remaining code (Fig. 4, "1 core Xeon" bar). However, once ported to the Cell/B.E. processor, runtime of the remaining code increases to more than 30% of the overall execution time (Fig. 4, "1 PPE, 8 SPEs" bar). The execution time of the part of the code that remains on the PPE slows down more than three times as compared to the execution time on the single core of the Intel Xeon chip, and the part of the code ported to the eight SPEs speeds up more than 12 times as compared to the Intel Xeon execution time. These observations hold true for different lattice sizes tested in this work. It is clear that PPE becomes the bottleneck in achieving any substantial performance increase beyond this point.

The QS20 IBM Cell blade consists of two Cell/B.E. processors mounted on the same board with a high-speed coherent interface between the two chips running at 20 GB/s in each direction [13]. Two ways to scale up the application on the dual-chip board are considered:

- Run the application on one PPE while offloading the 54 kernels to 16 SPEs. Since the performance is largely determined by the bandwidth between main memory and the SPEs, the memory

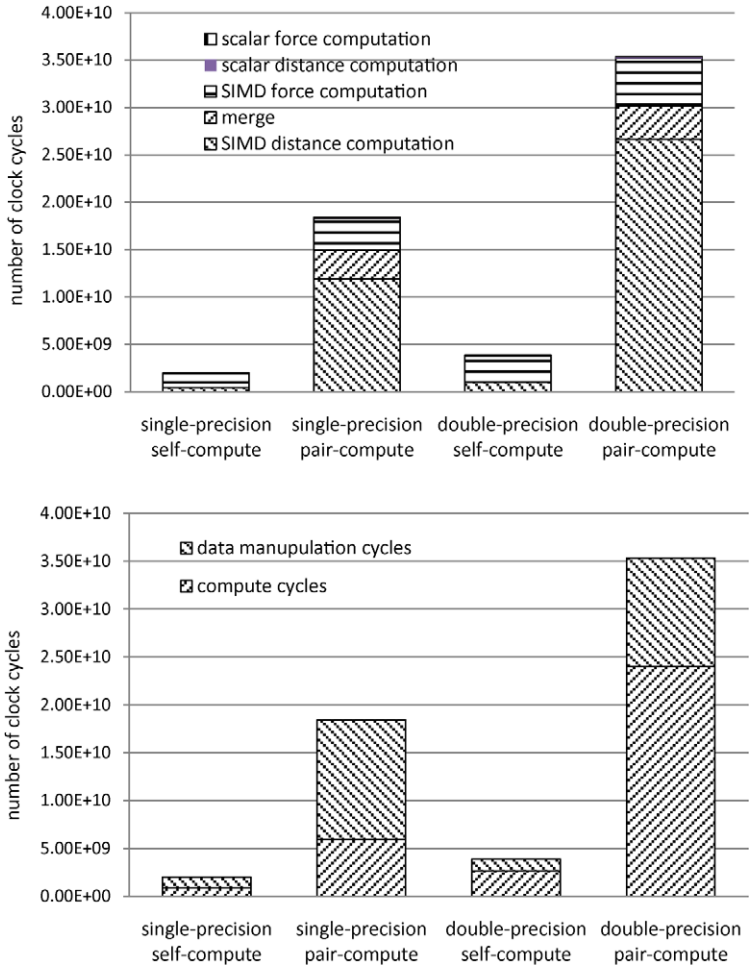

Fig. 2. Static analysis of the computing kernels performed with the help of the timing tool, spu_timing, provided with the Cell SDK. The time is taken from running the entire simulation on just one SPE, excluding DMA data transfer time. The top graph shows clock cycle distribution between different types of computations. SIMD distance and force computation bars correspond to processing four pairs of atoms at a time using SIMD instructions. If there are fewer than four pairs of atoms left, they are processed one by one using the original scalar code (scalar distance and force computation bars). The bottom graph shows clock cycle distribution between actual calculations and data manipulation operations.

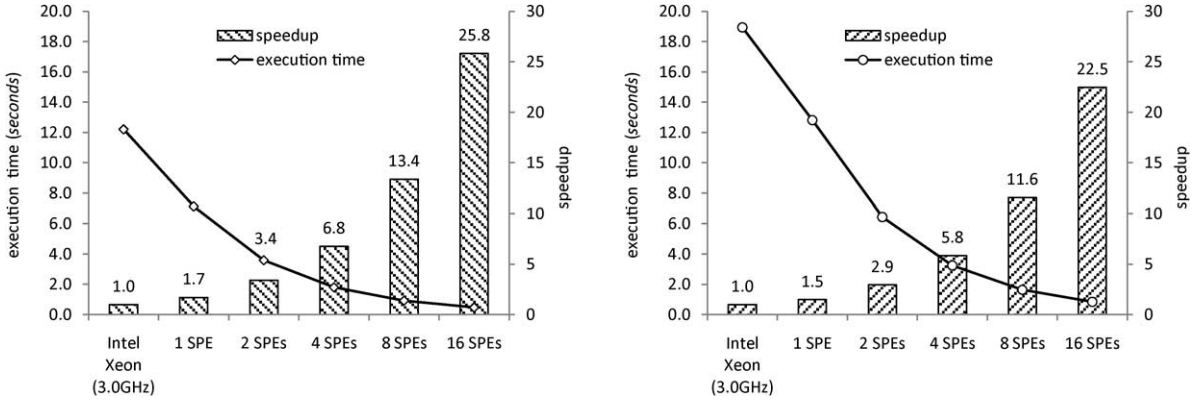

Fig. 3. Scaling and speedup of the Cell/B.E. NAMD force-field kernel implementation as compared to a 3.0 GHz Intel Xeon processor. Single-precision (left) and double-precision (right) kernels exhibit similar scaling behavior.

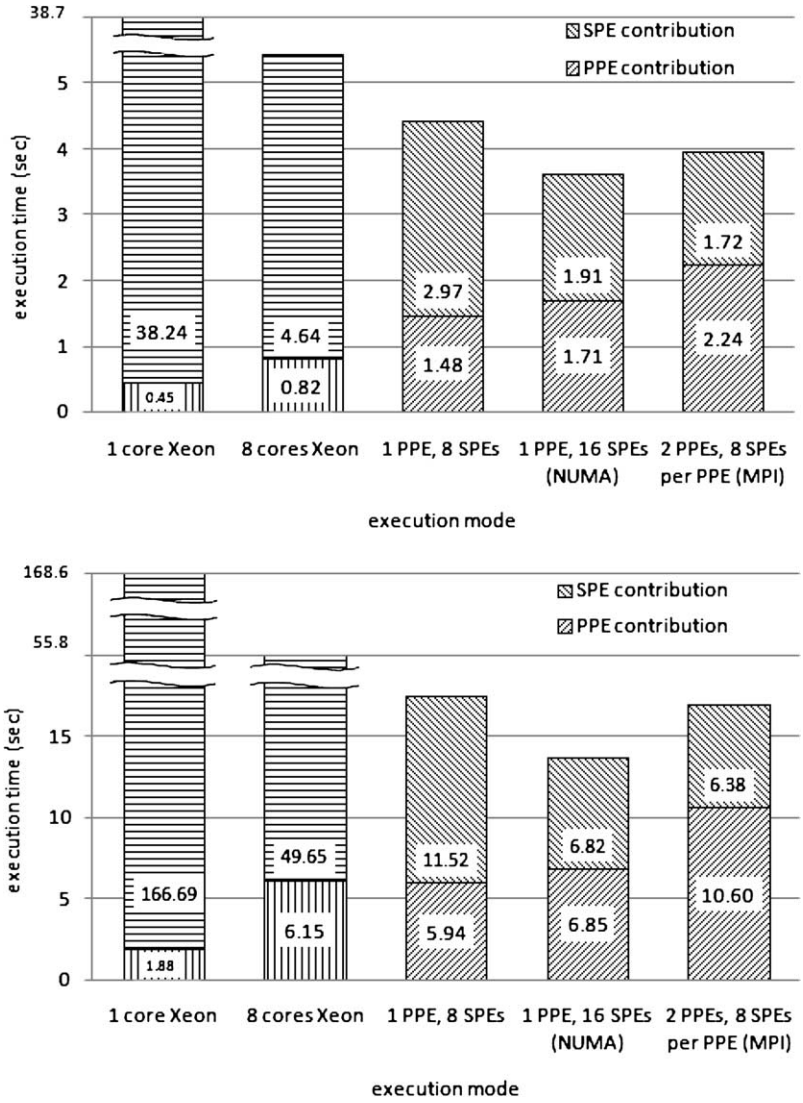

Fig. 4. Execution time of the PPE and SPE parts of the MILC code on the Cell blade for the lattice size of $8\times8\times16\times16$ (top) and $16\times16\times16\times16$ (bottom). Single- and multi-core Intel Xeon performance is provided for reference with the bottom part of the chart bars showing the execution time of the code that corresponds to the code that remains executed on the PPE and the top part of the chart bars showing the execution time of the code that is executed on the PPEs in the Cell/B.E. implementation.

is allocated alternatively between two Cell/B.E. processors to maximize the bandwidth. In this case, a slight increase in the time spent on the PPE and a small decrease of the time spent on the SPEs are observed, with overall runtime decreasing 18 to 22% for different lattice size (Fig. 4, "1 PPE, 16 SPEs (NUMA)" bar).

- Run the application on two PPEs as two MPI processes with each MPI process computing half of the full grid size. Each process runs on one PPE and offloads the computational kernels to its own eight SPEs. The two processes communicate using MPI. While the runtime for the SPE part of the code decreases (Fig. 4, "2 PPEs, 8 SPEs per PPE (MPI)" bar), the runtime of the remaining PPE part of the code increases to well over 50% of the overall execution time. Further profiling shows the added MPI communication overhead is about 50% of the overall PPE time. The overall performance of this implementation slightly increases when compared to the first execution schema.

These results are as expected: by going from 8 to 16 SPEs, the number of compute engines increases

by a factor of two, allowing the SPE-resident code to run faster by making available more memory bandwidth. However, spreading data between the memories of two Cell/B.E. processors did not increase the effective memory-to-SPEs local store bandwidth by a factor of two because in many instances SPEs from one Cell/B.E. chip access memory attached to the other Cell/B.E. chip, resulting in a higher latency and a lower bandwidth. An MPI implementation makes a better data localization per Cell/B.E. processor, but it is still not ideal. However, MPI itself runs quite slowly on the PPE, thus reducing the overall performance.

Figure 5 presents measurements for both floating-point operations per second used and data transfer bandwidth sustained by all 27 subroutines executed on eight PPEs. Only infrequently kernel performance of more than 10 GFLOPS is achieved with any of the kernels, which is just over 4% of the combined PPEs' peak performance. At the same time, only five kernels achieve less than 10 GB/s memory bandwidth

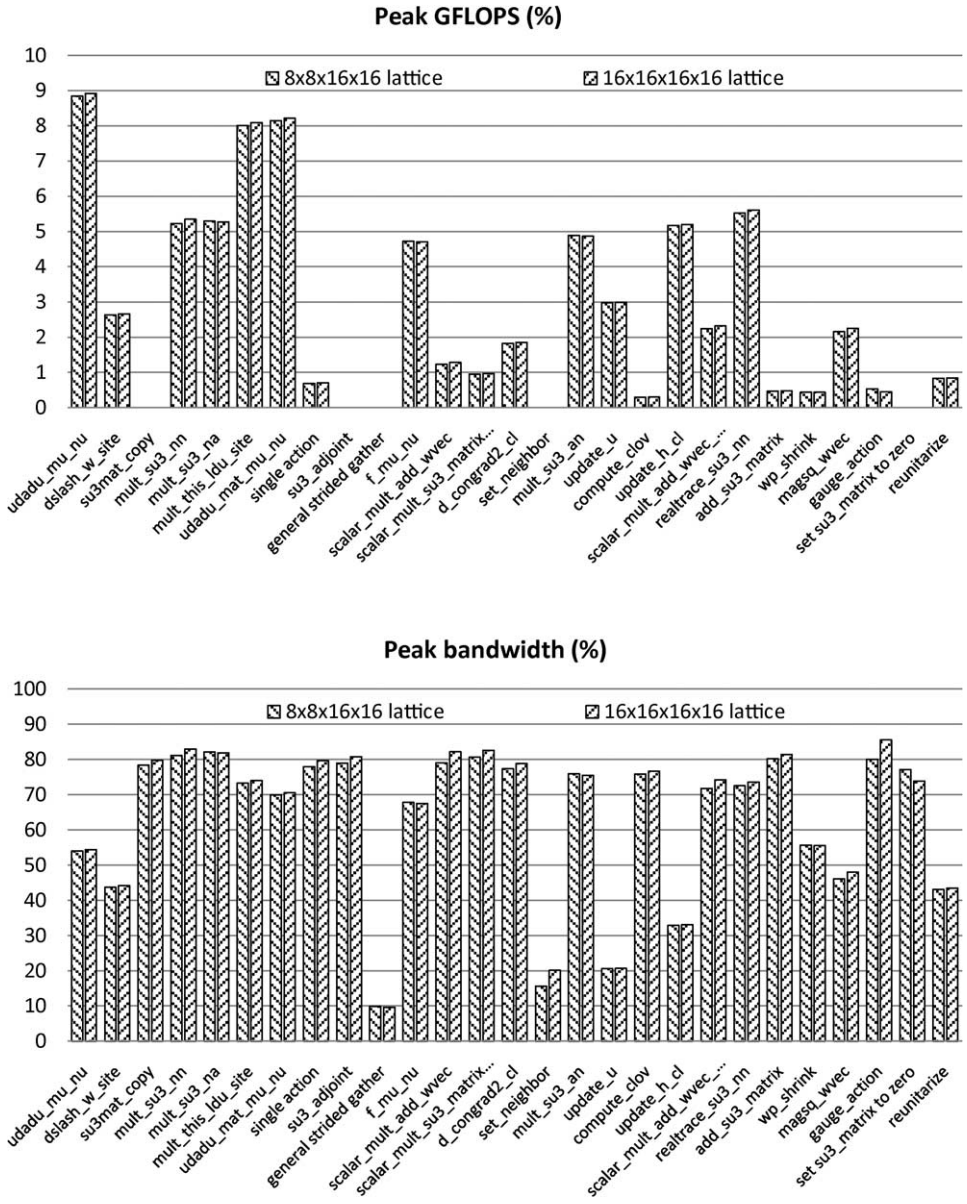

Fig. 5. Peak performance (FLOPS and bandwidth) utilization on the Cell/B.E. system for two lattice sizes. Results are consistent for both datasets. The larger dataset typically results in slightly better resource utilization.

utilization, and many of the kernels are sustaining a bandwidth close to 20 GB/s, which is 78% of the peak. To better explain these results, all kernels can be classified into three types. Kernels of the first type do actual floating-point calculations and usually result in non-zero FLOPs and high bandwidth utilization, e.g., mult_su3_nn. Kernels of the second type do not have any calculations, but move data around, e.g., su3mat_copy and set_su3_matrix_to_zero. For example su3mat_copy kernel, which is responsible for about 20% of the overall runtime on the Intel Xeon platform, is used to copy matrices in memory. When left on the PPE, it results in a significant slowdown because of the limited memory-to-PPE bandwidth. However, a speedup of $\sim 18\times$ as compared to the original Intel Xeon implementation is achieved by simply transferring the original data from source location in memory to local store in SPEs and then transferring it back to a new destination in main memory. Similarly, set_su3_matrix_to_zero (functionally analogous to the standard bzero subroutine) is implemented by transferring an array of zeroes from SPEs to main memory, resulting in a speedup of $\sim 41\times$ as compared to the original Intel Xeon implementation. Kernels of the third type do only integer calculations (and therefore do not show any FLOPS in Fig. 5) that are needed to compute neighboring address and construct MPI messages. Two kernels, general_strided_gather and set_neighbor, belong to this type. Bandwidth utilization for these kernels is low because only a small amount of data is transferred between the main memory and the SPEs and it is not aligned at 128-byte boundaries. However, by implementing these kernels on the SPEs, speedups of $16.6\times$ and $2.6\times$, respectively, are achieved when compared to the original Intel Xeon code.

Spray [21] reports 41 GFLOPS obtained on a single Cell/B.E. processor for the Wilson Dslash operator, which is functionally equivalent to MILC's dslash_w_site subroutine that achieves only about 5.4 GFLOPS in our implementation. Spray's implementation, however, is "scalar in nature, in the sense that the level of parallelization is limited to a single Cell blade. If multiple Cell blades are to be used, then the code would have to interleave off-node communications with on-processor calculations and communications" [21, p. 63]. Also, Spray's implementation is limited to datasets that fit into SPEs' LS, thus making it impractical for any real problems. MILC's dslash_w_site subroutine, on the other hand, consists of four kernels implemented on the SPE and interleaved with the MPI-based inter-node gather-scatter operations implemented on the PPE. Also, our Cell/B.E. implementation does not impose any limits on the dataset size. This largely explains the performance difference between Spray's implementation and the work reported here.

6.3. ERIs kernel

Two test cases are considered: a molecular system consisting of 10 water molecules (30 atoms in total) using cc-pVDZ basis set with only s-type functions taken into account and a molecular system composed of 64 hydrogen atoms arranged in a lattice using the STO-6G basis set. The first model is an example of a relatively uncontracted basis set, whereas the second model is an example of a highly contracted basis set. Figure 6 presents performance results for these models.

An obvious observation is that the speedup obtained for the model that uses a highly contracted basis set

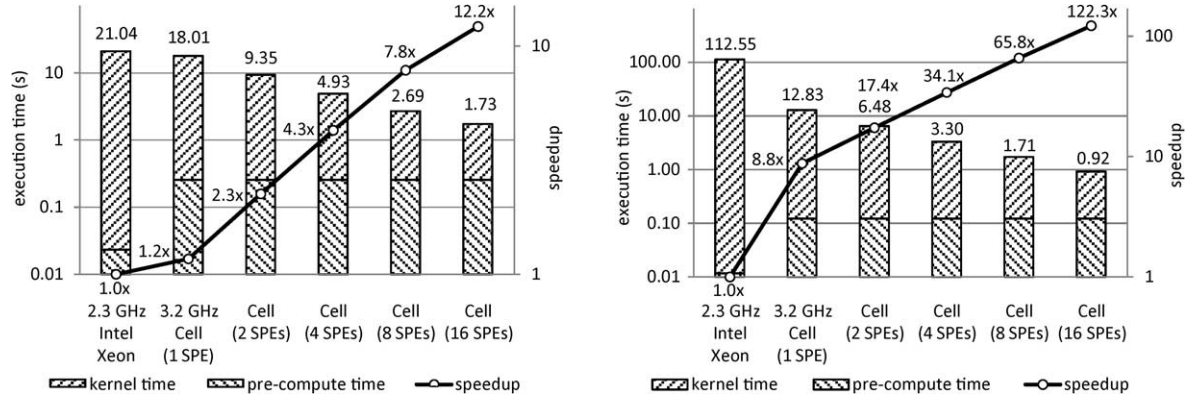

Fig. 6. Scaling (left Y axis) and speedup (right Y axis) of the Cell/B.E. two-electron repulsion integrals kernel implementation as compared to a 2.3 GHz Intel Xeon processor for model 1 (left) consisting of 30 atoms using cc-pVDZ basis set and model 2 (right) consisting of 64 atoms using STO-6G basis set.

is substantially higher than for the model that uses a relatively uncontracted basis set. The degree at which the basis set is contracted translates into the number of primitive integrals to be computed for each contracted integral. In terms of the code execution profile, this corresponds to the number of iterations of the innermost loop. In the case of the highly contracted basis set, the implementation in which the two innermost loops are fused is used, thus yielding a much better utilization of the SIMD engine. In the case of the relatively uncontracted basis set, the same efficiency cannot be achieved because there is no loop in which the number of iterations is a multiple of four. Thus, a relatively un-optimized version of the kernel is used.

The PPE pre-compute time for the first model is 0.25 s, whereas for the second model it is 0.12 s. The point is approaching, especially with the second model, when the PPE's performance is becoming a limiting factor in improving the overall code performance. One way to deal with this issue is to implement the pre-computing stage on the SPEs as well.

For both models linear scaling is observed when increasing the number of SPEs used to compute ERIs. This behavior is expected since the work is assigned to the SPEs based on the total number of integrals to be computed.

7. Conclusions and lessons learned

In this work, the use of the Cell/B.E. processor for implementing and running several scientific computing codes with both the single-precision and double-precision numerical types is explored. The results point out the potential of the Cell/B.E. processor to accelerate applications by up to two orders of magnitude when compared to mainstream microprocessors. These performance improvements, however, come with substantial data and code manipulations. When porting an application to the Cell/B.E. processor, two main issues arise:

- Uncovering application/algorithm parallelism that can be translated into an efficient multi-core SIMD implementation, and
- Removing obstacles in the way of approaching hardware limits.

While the issue of uncovering application/algorithm parallelism is beyond the scope of this paper, we would like to point out a few lessons and techniques that we used to approach the hardware limits. Table 3 lists main optimization techniques applied in each of the applications.

Lesson 1: Keep code on the PPE to a minimum. Because of the limited PPE-to-main memory bandwidth, any heavy-lifting calculations, or even memory operations, should not be executed on the PPE. In tests using the STREAM memory bandwidth benchmark, the PPE delivers only 1–1.5 GB/s of sustained bandwidth, whereas a single-core Intel Xeon chip sustains a transfer rate of over 3.5 GB/s. Coupled with a smaller cache size, code fragments that seem to take very little time on a conventional processor may easily become a substantial bottleneck on the PPE. As an example, in the case of MILC, memory copy and memory reset operations turned out to be much faster when implemented on the SPE (Fig. 5).

Table 3
Summary of optimization techniques applied in each of the applications

	NAMD	MILC	ERIs
Problem nature	N-body simulation	4D structured grid	O(N^4) compute-intensive problem
Performance limit	Computation bound	Memory bandwidth bound	Computation bound
SPE code optimizations	Explicit SIMD	Explicit SIMD	Explicit SIMD
	Branch elimination	Double buffering	Double buffering
	Partial loop unrolling	Memory structure padding	Array padding
		Memory affinity	Complete loop unrolling
		Elimination of capacity misses	Loop reordering
Load balancing	A pool-based dynamic task distribution schema. Each SPE works on separate tasks and the load balance is achieved due to the dynamic nature of the task assignment.	All SPEs work on the same task simultaneously and the work is evenly distributed between the SPEs. Many kernels, multiple calls to the kernels.	All SPEs work on the same task simultaneously and the work is evenly distributed between the SPEs. Single kernel, single call to the kernel.

Lesson 2: Know if your SPE kernel is bandwidth-limited or compute-limited. Knowing which hardware limits are stopping one from improving performance is very important in order to apply the appropriate optimization techniques. To demonstrate the point, compute-bound codes do not require optimizations to speedup data transfers whereas bandwidth-limited codes do not require optimizing computations because doing so will not remove the main bottleneck. In the case of the applications presented in this paper, NAMD and the ERIs kernel are compute-bound whereas the majority of MILC kernels are bandwidth-limited.

Lesson 3: One way to deal with bandwidth-limited kernels is to turn them into compute-bound kernels. This can be achieved by adding more work to the kernels. As an example, in the case of the ERIs kernel for the uncontracted basis sets, transferring pre-computed quantities from the main memory to SPE's local store becomes a bottleneck. However, these quantities can be computed directly on the SPEs as needed and the overall performance of the kernel improves. Thus, limited bandwidth can be traded for the unused FLOPs.

Lesson 4: Efficient use of the SPE's DMA engine requires proper data alignment. It is not only important to align data at 128-byte boundaries and pad it to the nearest 128 bytes to maximize the DMA bandwidth, but also to understand how strided memory access works. This was clearly demonstrated in the MILC implementation when accessing lattice site data in a strided manner. Thus, with an odd stride size (e.g., 1, 3, 5), the maximum bandwidth of 25.38 GB/s can be achieved, whereas with the stride size of 16, only 2.13 GB/s is achievable because only one out of 16 memory banks is used all the time [20].

Lesson 5: SIMD instructions require proper data structures and alignments. In order to make an efficient use of the SPE's SIMD instructions, operands need to be aligned at 16-byte boundaries and need to be arranged as vectors. In all considered applications this was the most time-consuming and difficult optimization to implement. Aligning data at the 16-byte boundaries frequently can be achieved by simply padding the corresponding data structures, with the obvious side effect of increasing memory size to hold the data. Assembling vectors for SIMD instructions is significantly more challenging since numerous clock cycles can be wasted to shuffle or accumulate the data before the required number of vector elements is assembled. This was especially true in the case of NAMD. This issue was dealt with by filling in the missing data with zeros and discarding the unneeded results at the end. In general, one needs to consider not only data structures, but also algorithm selection and possibly the entire application structure to vectorize the code efficiently.

Acknowledgments

This work was funded by National Science Foundation grants SCI 05-25308 and CHE-06-26354 and by the IBM Linux Technology Center. We acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation for the use of Cell Broadband Engine resources that have contributed to this research. We are also thankful to Dr. Paul Woodward from the University of Minnesota for providing access to the Cell blade system. We are thankful to Dr. Gregory Bauer from NCSA for useful comments on MILC performance evaluation. Special thanks to Jeremy Enos from NCSA's Innovative Systems Lab (ISL) for installing and administrating the Cell blade system at NCSA and to Trish Barker from NCSA's Office of Public Affairs for help in preparing this publication.

References

[1] A. Arevalo, R. Matinata, M. Pandian, E. Peri, K. Ruby, F. Thomas and C. Almond, *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, 1st edn, IBM Redbooks, Poughkeepsie, NY, 2008.

[2] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams and K. Yelick, The landscape of parallel computing research: A view from Berkeley, Technical Report No. UCB/EECS-2006-183.

[3] F. Belletti, G. Bilardi, M. Drochner, N. Eicker, Z. Fodor, D. Hierl, H. Kaldass, T. Lippert, T. Maurer, N. Meyer, A. Nobile, D. Pleiter, A. Schäfer, F. Schifano, H. Simma, S. Solbrig, T. Streuer, R. Tripiccione and T. Wettig, QCD on the Cell Broadband Engine, in: *Proc. XXV International Symposium on Lattice Field Theory*, Regensburg, Germany, 2007.

[4] G. Bilardi, A. Pietracaprina, G. Pucci, F. Schifano and R. Tripiccione, The potential of on-chip multiprocessing for QCD machines, *Lect. Notes Comput. Sci.* **3769** (2005), 386–397.

[5] S. Boys, Electronic wave functions. I. A general method of calculation for the stationary states of any molecular system, *Proc. R. Soc. London, Ser. A* **200** (1950), 542–554.

[6] G. De Fabritiis, Performance of the cell processor for biomolecular simulations, *Comp. Phys. Commun.* **176** (2007), 660–664.

[7] M. Dupuis, J. Rys and H. King, Evaluation of molecular integrals over Gaussian basis functions, *J. Chem. Phys.* **65** (1976), 111–116.

[8] S. Gottlieb, W. Liu, D. Toussaint, R. Renken and R. Sugar, Hybrid-molecular-dynamics algorithms for the numerical simulation of quantum chromodynamics, *Phys. Rev. D* **35** (1987), 2531–2542.

[9] T. Hayashi, H. Honda, Y. Inadomi, K. Inoue and K. Murakami, Implementation and evaluation of a molecular orbital calculation program on cell processor, 2006-HPC-107-(18), *Inform. Process. Soc. Japan (IPSJ)* **87** (2006), 103–108.

[10] J. Meredith, S. Alam and J. Vetter, Analysis of a computational biology simulation technique on emerging processing architectures, in: *Proc. IEEE Intl. Symposium on Parallel and Distributed Processing*, Long Beach, CA, USA, 2007.

[11] S. Motoki and A. Nakamura, Development of QCD code on a Cell Machine, in: *Proc. XXV International Symposium on Lattice Field Theory*, Regensburg, Germany, 2007.

[12] NAMD SPEC CPU2006 Benchmark Description, http://www.spec.org/cpu2006/Docs/444.namd.html.

[13] A. Nanda, J. Moulic, R. Hanson, G. Goldrian, M. Day, B. D'Amora and S. Kesavarapu, Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers, *IBM J. Res. & Dev.* **51** (2007), 573–582.

[14] S. Oliver, J. Prins, J. Derby and K. Vu, Porting the GROMACS molecular dynamics code to the cell processor, in: *Proc. IEEE Intl. Symposium on Parallel and Distributed Processing*, Long Beach, CA, USA, 2007.

[15] J. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. Skeel, L. Kale and K. Schulten, Scalable molecular dynamics with NAMD, *J. Comp. Chem.* **26** (2005), 1781–1802.

[16] D. Pleiter, Lattice QCD on the Cell BE, in: *Proc. Power Architecture Developer Conference*, Austin, TX, USA, 2007.

[17] T. Ramdas, G.K. Egan, D. Abramson and K.K. Baldridge, Uniting extrinsic vectorization and shell structure for efficient SIMD evaluation of Electron Repulsion Integral, *Chem. Phys.* **349** (2008), 147–157.

[18] M. Schmidt, K. Baldridge, J. Boatz, S. Elbert, M. Gordon, J. Jensen, S. Koseki, N. Matsunaga, K. Nguyen, S. Su, T. Windus, M. Dupuis and J. Montgomery Jr., General atomic and molecular electronic structure system, *J. Comp. Chem.* **14** (1993), 1347–1363.

[19] G. Shi and V. Kindratenko, Implementation of NAMD molecular dynamics non-bonded force-field on the Cell Broadband Engine processor, in: *Proc. IEEE Intl. Symposium on Parallel and Distributed Processing*, Miami, FL, USA, 2008.

[20] G. Shi, V. Kindratenko and S. Gottlieb, Cell processor implementation of a MILC lattice QCD application, in: *Proc. The XXVI International Symposium on Lattice Field Theory*, Williamsburg, VA, USA, 2008.

[21] J. Spray, Lattice QCD on the Cell Processor, MS Thesis, The University of Edinburgh, 2007.

[22] The MIMD Lattice Computation (MILC) Collaboration, http://www.physics.utah.edu/∼detar/milc/.

[23] I. Ufimtsev and T. Martinez, Quantum chemistry on graphical processing units. 1. Strategies for two-electron integral evaluation, *J. Chem. Theor. Comput.* **4** (2008), 222–231.

[24] M. Wolf, Efficient linear algebra related to lattice QCD on Cell Broadband Engines, Technical Report, Ruprecht-Karls-University, Heidelberg, 2007.